Customized OS support for data-processing

Jana Giceva, Gerd Zellweger, Gustavo Alonso, Timothy Roscoe
Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
name.surname@inf.ethz.ch

ABSTRACT

For decades, database engines have found the generic interfaces offered by the operating systems at odds with the need for efficient utilization of hardware resources. As a result, most engines circumvent the OS and manage hardware directly. With the growing complexity and heterogeneity of modern hardware, database engines are now facing a steep increase in the complexity they must absorb to achieve good performance. Taking advantage of recent proposals in operating system design, such as multi-kernels, in this paper we explore the development of a light weight OS kernel tailored for data processing and discuss its benefits for simplifying the design and improving the performance of data management systems.

1. INTRODUCTION

Database engines running on today’s operating systems suffer from a number of performance related problems primarily caused by generic OS policies:

First, when data processing runs concurrently with other applications on the same machine, the default OS policies for scheduling and resource allocation often cause performance degradation [10, 24] and inefficiencies in resource utilization [15, 37].

Second, even when running in isolation, databases pay a lot of attention when managing the resources to avoid the penalties from default OS policies [27, 29, 42]. Examples include pinning threads to cores, allocating memory from a particular NUMA node, or pinning pages to avoid swapping, etc. [5, 32, 35, 45]. Many of these optimizations are tailored to a particular hardware architecture and are, thus, not easily portable to other platforms [31, 33, 38, 47].

Third, even when the optimizations deliver the desired performance and predictability properties, they are often fragile as they rely on a particular implementation of specific OS kernel mechanisms and policies (e.g., HyPer [39] relies on efficient OS-assisted snapshotting). As a consequence, any changes to the operating system can cause performance penalties.

In this paper, we propose a solution that tailors the OS stack, and in particular the OS kernel, to the needs of the database system or its workloads (§ 3). It revisits the decades-old problem of OS support for databases [18, 49] in the context of modern hardware. The solution we advocate relies on recent advancements in operating systems which enable systems to run a specialized OS kernel on a subset of the resources of a given machine (§ 2.2). In this way, the database engine gets considerably more control over the full system stack, which can then be tuned to achieve better performance and provide stronger guarantees.

As example take the default OS policy for scheduling threads on cores. The OS main principle is multiplexing resources among all applications currently active in the system. With its limited to non-existing knowledge of the requirements of the applications, it migrates, preempts, and interrupts threads on various cores trying to optimize certain system-wide OS metrics [7]. While existing libraries allow to override thread migration by pinning threads to cores, it still does not stop the OS from preempting the thread and scheduling another activity on the same core. Such disruptions at inconvenient times can be detrimental to both performance and predictability for data processing systems [16, 20]. A customized OS kernel can address this problem by providing support for dedicated cores, which we discuss in more detail in § 5. Other common problems and examples are provided in § 4.

1.1 Motivating example

Modern hardware provides many opportunities for resource sharing. Figure 1 illustrates a concrete example using Intel’s SandyBridge many-core machine.

The effects of resource sharing on the performance of systems have been extensively studied [16, 31, 59]. This so-called noise as perceived by the application can be (1) external, e.g., from the OS or another application or (2) internal, i.e., from within the database system itself as a result of multi-tenancy or nested-parallelism (e.g. handling multiple parallel queries concurrently).

When designing suitable core scheduling policies, databases have
to absorb additional complexity and understand (1) the model of the underlying architecture; (2) the workload characteristics and the resource requirements of the operations to be scheduled; as well as (3) the behaviour of the relational operators when executed in a noisy environment, i.e., their sensitivity when sharing resources and how that impacts their performance.

In Figure 2, we show the observed slowdown for a relational operator when collocated with another operator such that they share a particular hardware resource. The numbers are normalized to the performance of the individual operators when executed in isolation. The results indicate that the observed slowdown can vary significantly depending on both (1) the type of resource being shared, and (2) the noise, i.e., the properties of the partner operator. However, when the noise is external, neither the sophisticated machine model nor the workload characterization is of help as the database system is not aware of the system state and current utilization of the hardware resources. Similarly, if a data processing system does not absorb the above mentioned complexity and relies on the standard runtime or operating system, then the impact of the resource sharing is further aggravated, as we will show in § 5.3. The challenges of data processing were also explored by Porobic et al. in the context of OLTP workloads [43].

A customized OS kernel can address such issues and give performance isolation guarantees by providing a parallel task based execution mechanism. We provide more details in § 4.

2. BACKGROUND

2.1 Challenges for DB on modern hardware
Existing data-processing systems have two alternatives for their implementation: (1) they rely on existing runtime or OS mechanisms for memory management, synchronization and thread scheduling (e.g. POSIX, OpenMP and JVM); or (2) implement optimized data structures and primitives themselves, tuned for a particular task on a given hardware (e.g., suitable synchronization primitives as required by the chosen concurrency control mechanism [36]).

The first option is usually chosen by engines that are willing to trade-off performance for easier maintenance and portability across different platforms and hardware architectures (e.g., Spark [56], GraphX [54], GreenMarl [22]). The second option is preferred by data-processing systems that optimize heavily for performance (e.g., HyPer [26], MonetDB [6], Shore-MT [23]). However, it is well understood that these individual optimizations come at a cost of portability, especially as hardware becomes more diverse [41]. Simply absorbing all the complexity of the underlying hardware is not a viable solution in the long run. Instead of replicating the overhead of hardware-tuned implementations in each data-processing system, a better approach is to extract the most common primitives and mechanisms and support them as part of the operating system.

2.2 Advances in operating systems
Commodity operating systems like Windows, BSD and Linux aim to satisfy the requirements of a wide range of applications. There are, however, several application domains whose requirements cannot be supported well in general-purpose operating systems: (1) High performance computing (HPC) systems are very sensitive to OS noise. Such noise, aggravated by the scale at which these systems run, results in severe performance problems [21]. Thus, supercomputing systems have started using customized lightweight kernels [13, 25, 46]; (2) Real-time systems controlling or monitoring physical equipment and infrastructure typically have hard real-time requirements – hence, the development of real-time OSes.

Databases face similar problems as they only require a subset of the services, general purpose mechanisms, and primitives that a general-purpose OS provides. However, most data-processing systems run on commodity machines or on high-end manycore servers, and often share the machine with other applications or systems that may have different requirements. Therefore, even though compelling, such customized lightweight operating systems were not a viable option in the past because of the already mentioned monolithic architecture of conventional operating systems.

Some new OSes are based on a multikernel design [3] which runs a separate kernel on every core [50, 52]. It allows for greater flexibility as now even the kernel-space mechanisms and primitives can be tailored to the application’s needs. In parallel, the HPC community is already exploring this design by having the light-weight kernels (LWK) running alongside full-weight kernels (FWK) like Linux inside the same system [12,48,53]. It is the flexibility that the multi-kernel provides which would enable optimized lightweight OS-support for database systems to co-exist in the same system with other general-purpose kernels. Additionally, it is a good fit when addressing hardware heterogeneity and customizing the OS support for different computational resources. We illustrate these benefits of a multi-kernel in Figure 3.
3. DESIGN

We base our specialized OS support for data-processing systems on the following three design principles:

1. Control-/compute-plane framework, which allows the co-existence of multiple different kernels. Systems which require more traditional OS support can still be executed on the full-weight OS (control-plane), and delegate the performance-critical portions of their functionality (i.e., data processing) to the customized light-weight OS (compute-plane).

2. Specialization across the whole system stack. We distinguish two forms of specialization: (1) customization based on the application requirements, and (2) hardware tuning in the system stack for a variety of machine architectures.

3. Leverage novel OS services that allow direct access to hardware from user-space via better hardware abstractions.

3.1 Customized OS stack overview

The architecture of our proposed OS support for database systems is shown in Figure 4. The OS stack is separated into several components, based on whether the selected services can be implemented in user- or kernel-space, as well as whether they should figure as part of the general purpose OS library or application-tuned runtime system (RTS).

The customized lightweight kernel (i) contains mechanisms and services that can not be otherwise implemented in user-space. In contrast to Exokernels, we use kernel-level specialization to directly interact with the global scheduling policy of the operating system. This functionality is typically not exposed to the application as it requires a global view of the system state. We discuss the scheduling in our system in § 5.2; and (ii) is lightweight which, in principle, means less overhead (by removing unnecessary jitter and functionality which is not required by the database), better scalability on rack-scale machines, and can result in more predictable performance (no undesired interrupts, handling of expensive system calls, preemption, etc.). We describe its mechanisms and services in detail in § 4.1.

The library OS, contains hardware and application-tuned implementation of OS services and mechanisms for data processing. The goal is to cover the basic features in user-space that would allow systems which primarily care for portability still have access to high performing abstractions tuned for data processing, while at the same time provide flexibility for the systems optimizing for performance to override them with a customized implementation.

Finally, the OS runtime system provides additional functionality and higher-level interface for highly tuned memory management primitives and a task-based scheduling mechanism, leveraging the support provided by the lightweight kernel.

In the rest of the paper, we focus the discussion on the mechanisms we selected to be supported by the lightweight kernel.

4. CUSTOMIZED LWK

We classify the kernel support into several categories depending on the offered functionality: (1) management of computational resources, (2) memory management, and (3) providing more transparent access and management of various hardware devices.

4.1 Managing CPU resources

Dedicated CPU resources. Thread and data-placement play a great role in the performance of database systems. Therefore, it has been common practice not to rely on the operating system to deploy and manage threads (due to the OS’ thread migration policies). Instead, systems use libraries like libnuma to pin threads to a particular core. Moreover, they can also specify the priority level for scheduling a certain group of threads. Unfortunately, none of these provides guarantees for isolation, i.e., that a given thread will get dedicated access to the CPU core without being interrupted or context-switched. Although there are cases in which such a guarantee is not essential, there are many others in which the consequences of sharing a core can be detrimental to the performance and predictability of the application. In COD [16] we have demonstrated the severe effects that CPU sharing even on one of the cores can have on a CPU-intensive task, like a heavily optimized scan operator [51], with performance drops of as much as 77%.

Run-to-completion tasks. Even in cases where preemption and CPU-sharing is acceptable, or needed for more efficient resource utilization and consolidation, such an event and interruption must happen at convenient times. Otherwise, as shown in Callisto [20], the slowdown can be quite significant ranging between 30-100% for synchronization-heavy applications. The slowdown is usually due to preempting or interrupting the thread execution logic at inconvenient times (e.g., while holding a latch or working on a performance critical section). Therefore, we introduce a kernel abstraction for task-based execution, where the OS will not interrupt and/or preempt the thread and instead run tasks to completion.

Co-scheduling in noisy environment. If the CPU resources within a hardware-island are scheduled well, the database can leverage the resource sharing potential – either by collocating communicating threads, or by using the caches and data-locality for data-intensive processing. However, if managed poorly in a noisy multiprogramming environment, the performance of the system can be significantly impacted.

The results in Figure 2 show that sometimes even sharing the last-level cache and local DRAM bandwidth can slow down the execution of an operator by almost 60% (hashjoin). Hence, simply controlling each core individually is not enough to avoid such scenarios, and more advanced scheduling mechanisms are needed.

Therefore, the proposed LWK has scheduling support that makes sure that, when necessary, all cores used by one job, are coordinated and scheduled as a single unit. They could be assigned as a dedicated resource to a particular parallel operation, or co-scheduled among several such jobs.

Other directions. Recent advances in operating systems also allow for fast core booting [57], which can be used for more energy-efficient resource utilization, or even for handling the upcoming challenges of dark-silicon [9]. This is an aspect that we will explore in future work.

4.2 Memory management

The effects of non-uniform memory accesses, bandwidth and their implication on synchronization and memory movement has been extensively studied [35,43,45]. Similar analysis and optimizations
were done for the role of the TLB and reducing the costly overhead of virtual memory bottleneck [28, 55]. The virtual memory system is becoming a bottleneck for main-memory data processing. This inspired recent proposals such as direct segments [2] which allow circumventing the physical-to-virtual address translation for large regions of memory. Furthermore, OS swapping policies do not take into account application specific knowledge about the content of pages, something which has been shown to negatively impact performance of database systems [17]. Such effects are likely to become more pronounced with the introduction of NVRAM [29].

Unfortunately, the conventional operating system APIs for manipulating a process’ address space still try to shield the application from all this complexity – hence limiting the opportunity for applications to directly manage memory and their own page-tables or use self-paging to swap their data out to disk or NVRAM. Recent proposals in OS memory systems provide greater flexibility to applications by exposing both physical and virtual memory directly to application and allowing them to safely construct their own page-tables [11]. Such systems avoid the potentially sub-optimal global policies enforced by the OS.

Finally, database systems which would like to support OLAP workloads by taking snapshots of the OLTP datastore (inspired by tables [11]). Such systems avoid the potentially sub-optimal global policies enforced by the OS.

Other directions. In the future, the kernel should also expose more insightful models of the machine, as well as a better overview of the current utilization of the shared resources based on the current system-state.

5. PROTOTYPE

Our current implementation is an extension of the Barrelfish [50] operating system. We have implemented a light-weight kernel to provide only the necessary services needed for a database system. The light-weight kernel can be spawned on either sockets or individual cores. It currently offers the following features:

1. Task-based scheduling as opposed to the standard thread abstraction provided by a general-purpose kernel. The difference of how the kernel treats tasks and threads is that the task-based scheduler will never preempt or interrupt a running task.
2. Support to form and schedule groups of tasks (called a parallel task – ptask) that are strictly executed together on the same socket.
3. The dedicated CPU mechanism uses the task based abstraction – hence no interrupts or preemption – but also constrains the scheduler from allocating any other task on the queue for that particular CPU.
4. The memory and hardware access services described in the previous section are inherited from the existing support in Barrelfish.

5.1 Interface to database systems

The functionality provided by the LWK is exposed to runtimes through a library OS (see Figure 5) which allows data processing systems to create tasks, compose them to form parallel tasks, and enqueue parallel tasks in the system to be executed on the customized light-weight kernel (compute-plane). This is done by using a system call interface on the control plane which is still running a full-weight Barrelfish kernel. The compute plane kernel instances will then dequeue the ptasks and make sure they are scheduled with respect to the spatial as well as temporal co-scheduling constraints discussed in § 4.1. The control plane can wait for the completion of ptasks or abort their execution after they have been enqueued.

Further, the library OS abstracts the highly optimized message passing mechanisms integrated in Barrelfish to provide communication among tasks inside a ptask, but also between the control and compute-plane.

5.2 Example use-cases

Next, we provide several concrete use-cases of data processing engines which show how one can leverage the proposed kernel-based services. We focus on the services for managing CPU resources.

Use of a dedicated CPU. There are many scenarios where a part of a database system may need to run in isolation due to hard SLA guarantees. In such scenarios a mechanisms providing a dedicated CPU resource is of obvious use. Another example are operator-centric data processing systems, which have attracted attention in recent years (DataPath [1], QPipe [19], SharedDB [14], stream-processing, etc.). They are characterized by long-lasting query

```c
struct task { task_fn fun, void* arg, ... }
struct ptask { struct task* tasks, size_t count, ... }
bas_task_create(count) → struct task*
bas_ptask_create(tasks) → struct ptask*
bas_ptask_enqueue(ptask, locality, ...)
bas_wait(ptask)
bas_abort(ptask)
```

Figure 5: The parallel task-based execution API.
plans and hence have long-running operators. Some of the operators in a blocking operator pipeline (or with relatively low resource requirements) can easily be consolidated on a smaller set of cores [15]. There are, however, non-blocking pipelines with heavy CPU-intensive operators which are continuously active (e.g., a Crescendo scan [51] in SharedDB, or an optimized filter in a streaming engine). Their performance can be significantly reduced when sharing the CPU with another CPU-intensive task, as we have shown in the evaluation of COD [16]. Due to the nature of pipeline-based systems, impacting the performance of one such critical operator will negatively affect the performance of the whole system.

**OS support for task-based scheduling.** Apart from the obvious benefits for synchronization-heavy workloads and operations (as discussed in the previous section), the kernel support for task-based scheduling can be also leveraged by (1) task-based scheduling approaches recently introduced in SAP Hana [44] or HyPer [32] (i.e., the morsel-driven parallelism), which is important not to be context-switched or interrupted while processing a highly optimized operation on a morsel of data in order to keep both the instruction and data cache locality intact; (2) other optimized data-processing operators that rely on having the hot data fit in the L1/L2 caches, especially if the data is accessed randomly and any extra trip to DRAM is prohibitively expensive. Li et al. have shown that the total cost of a context switch can increase by almost three orders of magnitude for data-sensitive applications [34].

**NUMA-aware scheduling of parallel tasks** can be leveraged by any data-processing system internally using nested parallelism. For databases, in particular, in the case of bushy- and intra-operator parallelism, i.e., when executing multiple parallel queries concurrently. For instance, the execution engine needs to concurrently schedule multiple parallel relational operators (e.g., a parallel hash join or aggregation). The *ptask scheduling* policy makes sure that the threads belonging to the same *ptask* are placed on the same socket, and can benefit from constructive sharing of the last level cache. It also removes the danger of destructive resource sharing as the ones observed in Figure 2.

### 5.3 Initial evaluation

We evaluated the NUMA-aware scheduling policy adopted by our *parallel task scheduling* mechanism using the GreenMarl graph-processing system implemented on top of OpenMP on an Intel SandyBridge machine using the GreenMarl [22] algorithms for Page Rank, Hop Distance and Single Source Shortest Path on the Twitter dataset [30]. The two heat-maps in Figure 6 show the pairwise execution for each of the algorithms with different scheduling strategies in a noisy environment (i.e., paired with a second OpenMP runtime, running another algorithm) on four sockets.

In (a) we use the default Linux and OpenMP scheduling and measure the runtimes for two algorithms running together on four sockets. The numbers show the slowdown when compared against running one of the algorithms alone on just two sockets. In (b) we use a NUMA-aware scheduling policy by giving each OpenMP runtime two predetermined sockets, and compare it to the baseline of running a single OpenMP instance on two sockets inside the system.

For Figure 6a, we observe that for seven out of nine combinations, there is a noticeable slowdown with factors up to 2.5x compared to running the algorithms in isolation. We attribute these effects to the runtime oblivious assignment of hardware threads by the OS. The two OpenMP programs executing these algorithms are now open to memory bandwidth contention, L3 cache interference (by distributing threads from both programs on all sockets), thread migrations, the Linux first-touch policy for memory allocation, etc.

![Figure 6](image-url)  
*Figure 6: Slowdown in noisy environment compared to run in isolation of PageRank (PR), HopDistance (HD) and SingleSourceShortestPath (SSSP): standard vs. NUMA-aware scheduling.*

In this paper, we primarily focused on the kernel itself. However, we are also working on the corresponding library OS and runtime for data-processing systems. We would also like to explore if there is a need to specialize further, for instance, to provide other LWK feature-sets for OLAP and OLTP workloads.

Furthermore, the use of lightweight kernels is not just suitable for traditional multi-core machines but the same concept can be applied for better integration of heterogeneous systems with asymmetric performance characteristics, including hardware accelerators (e.g., XeonPhi). In addition, the distributed nature of a multi-kernel makes it a suitable candidate for rack-scale machines where scalability on all layers is key to the overall system performance.

### 6. DISCUSSION AND CONCLUSION

We presented the design of a light-weight kernel tailored to the needs of data-processing systems, and demonstrated that with its customized policies and mechanisms it can provide better performance in noisy environments. Additionally, the kernel exposes alternative interfaces and optimizations which were previously not possible with general-purpose operating systems.

In this paper, we primarily focused on the kernel itself. However, we are also working on the corresponding library OS and runtime for data-processing systems. We would also like to explore if there is a need to specialize further, for instance, to provide other LWK feature-sets for OLAP and OLTP workloads.

Furthermore, the use of lightweight kernels is not just suitable for traditional multi-core machines but the same concept can be applied for better integration of heterogeneous systems with asymmetric performance characteristics, including hardware accelerators (e.g., XeonPhi). In addition, the distributed nature of a multi-kernel makes it a suitable candidate for rack-scale machines where scalability on all layers is key to the overall system performance.

### 7. REFERENCES


